10.2 Object-Oriented Modelling of Our Problem Domain

In the previous section, we said that a system is a collection of entities that interact with each other over time. In this section, we will explore what data should be a part of our problem domain—a food delivery system—and how that data might change over time. We'll introduce an object-oriented approach to modelling this data in Python, using both data classes and general classes to represent different entities.

One thing to keep in mind as we proceed through this section (and the rest of the chapter) is that just like in the "real world", the scope of our problem domain is not fixed and can change over time. We are interested in the minimum set of data needed for our system to be meaningful, keeping the scope small at first with the potential to expand over time. Throughout this section, we'll point out places where we make *simplifying assumptions* that reduce the complexity of our system, which can serve as potential avenues for your own independent explorations after working through this chapter.

Entities in a food delivery system

A good first step in modelling our problem domain is to identify the relevant entities in the domain. Here is our initial description of *Hercules* from the previous section:

> Consider a person or household doing a self-quarantine during the pandemic. One of the main logistical challenges they have to face is how to arrange for food during their quarantine. To help address this need, you have founded *Hercules Ltd.*, a non-profit organization that allows people under quarantine to order groceries and meals from grocery stores and restaurants, and arranges for couriers to make deliveries right to their front doors.

We use two strategies for picking out relevant entities from an English description like this one:

1. Identify different roles that people/groups play in the domain. Each "role" is likely an entity: e.g., *customer*, *courier*, and *restaurant* are three distinct roles in the system.
2. Identify a bundle of data that makes sense as a logical unit. Each "bundle" is likely an entity: e.g., an *order* is a bundle of related information about a user's food request.

In an object-oriented design, a standard approach is to create a class to represent each of these entities. Should we make a data class or a general class for each one? There are no easy answers to this question, but a good strategy to use is to *start* with a data class, since data classes are easier to create, and turn it into a general class if we need a more complex design (e.g., to add methods, including the initializer, or mark attributes as private).

```python
@dataclass
class Restaurant:
    """A place that serves food."""

@dataclass
class Customer:
    """A person who orders food."""

@dataclass
class Courier:
    """A person who delivers food orders from restaurants to customers."""

@dataclass
class Order:
    """A food order from a customer."""
```

Once we have identified the classes representing the entities in the system, we now dive into the details of the system to identify appropriate *attributes* for each of these data classes. We'll discuss our process for two of these data classes in this section, and leave the other two to lecture this week.

Designing the `Restaurant` *data class*

Let us consider how we might design a restaurant data class. What would a restaurant need to have stored as data? It is useful to envision how a user might interact with the app. A user might want to browse a list of restaurants available, and so we need a way to identify each restaurant: its *name*. After selecting a restaurant, a user needs to see what food is available to order, so we need to store a *food menu* for each restaurant. Finally, couriers need to know where restaurants are in order to pick up food orders, and so we need to store a *location* for each restaurant.

Each of these three pieces of information—restaurant name, food menu, and location—are appropriate *attributes* for the restaurant. Now we have to decide what data types to use to represent this data. You have much practice doing this, stretching back to all the way to the beginning of this course! Yet as we'll see, there are design decisions to be made even when choosing individual attributes.

- The restaurant *name* is fairly straightforward: we'll use a `str` to represent it.
- The restaurant *menu* has a few different options. For this section, we'll use a `dict` that maps the names of dishes (`str`s) to their price (`float`s).

- There are many ways to represent a restaurant's location. For example, we could store its address, as a `str`. Or we could improve the precision of our data and store the latitude and longitude (a tuple of `floats`), which would be useful for displaying restaurants on maps.

  For now, we'll store both address and latitude/longitude information for each restaurant. It may be that both representations are useful, and should be stored by our application.

```python
@dataclass
class Restaurant:
    """A place that serves food.

    Instance Attributes:
      - name: the name of the restaurant
      - address: the address of the restaurant
      - menu: the menu of the restaurant with the name of the dish mapping to
        the price
      - location: the location of the restaurant as (latitude, longitude)

    Representation Invariants:
      - all(self.menu[item] >= 0 for item in self.menu)
      - -90 <= self.location[0] <= 90
      - -180 <= self.location[1] <= 180
    """
    name: str
    address: str
    menu: dict[str, float]
    location: tuple[float, float]
```

There is one other subtlety with this design before we move on. The menu is a compound data type, and we chose to represent it using one of Python's built-in data structures. But another approach would have been to create a completely separate `Menu` data class. That is certainly a viable option, but we were wary of falling into the trap of creating too many classes in our simulation. Each new class we create introduces a little more complexity into our program, and for a relatively simple class for a menu, we did not think this additional complexity was worth it.

On the flip side, we could have used a dictionary to represent a restaurant instead of a `Restaurant` data class. This would have reduced on area of complexity (the number of classes to keep track of), but introduced another (the "valid" keys of a dictionary used to represent a restaurant). There is always a trade-off in design, and when evaluating trade-offs one should never forget cognitive load on the programmer.

Designing the `Order` data class

Now let's discuss a data class that's a bit more abstract: a single order. An order must track the *customer* who placed the order, the *restaurant* where the food is being ordered from, and the *food items* that are being ordered. We can also imagine that an order should have an associated courier who has been assigned to deliver the order. We'll also keep track of when the order was created, and when the order is completed.

There's one subtlety with two of these attributes: the associated courier and the time when the order is completed might only be assigned values after the order has been created. So we use a default value `None` to assign to these two instance attributes when an `Order` is first created. We could implement this by converting the data class to a general class and writing our own `__init__` method, but instead we'll take advantage of a new feature with data classes: the ability to specify default values for an instance attribute after the type annotation.

```
@dataclass
class Order:
    """A food order from a customer.

    Attributes:
      - customer: the name of the customer who placed this order
      - restaurant: the name of the restaurant the order is placed for
      - food_items: a mapping from names of food to the quantity being
        ordered
      - start_time: the time the order was placed
      - courier: the courier assigned to this order (initially None)
      - end_time: the time the order was completed by the courier (initially
        None)

    Representation Invariants:
      - self.food_items != []
      - all(self.food_items[i][1] > 0 for i in range(len(self.food_items)))
    """
    customer: Customer
    restaurant: Restaurant
    food_items: dict[str, int]
    start_time: datetime.datetime
    courier: Optional[Courier] = None
    end_time: Optional[datetime.datetime] = None
```

The line `courier: Optional[Courier] = None` is how we define an instance attribute `Courier` with a default value of `None`. The type annotation `Optional[Courier]` means that this attribute can either be `None` or a `Courier` instance. Similarly, the `end_time` attribute must be either `None` (its initial value) or a `datetime.datetime` value.

Here is how we could use this class (note that `Customer` is currently an empty data class, and so is instantiated simply as `Customer()`):

```
>>> david = Customer()
>>> mcdonalds = Restaurant(name='McDonalds', address='160 Spadina Ave',
...                        menu={'fries': 4.5}, location=(43.649, -79.397))
>>> order = Order(customer=david, restaurant=mcdonalds,
...               food_items={'fries': 10},
...               start_time=datetime.datetime(2020, 11, 5, 11, 30))

>>> order.courier is None  # Illustrating default values
True
>>> order.end_time is None
True
```

Class composition

Just as we saw earlier in the course that built-in collection types like lists can be nested within each other, classes can also be "nested" within each other through their instance attributes. Our above `Order` data class has attributes which are instances of other classes we have defined (`Customer`, `Restaurant`, and `Courier`).

The relationship between `Order` and these other classes is called **class composition**, and is a fundamental to object-oriented design. When we create classes for a computational model, these classes don't exist in isolation. They can interact with each other in several ways, one of which is composition. We use class composition to represent a "has a" relationship between two classes (we say that "an `Order` has a `Customer`").[1]

[1] This is in contrast to inheritance, which defines an "is a" relationships between two classes, e.g. "`Stack1` is a `Stack`".